IT - 314
SOFTWARE ENGINEERING

Lab - 09: Mutation Testing

Group No - 11

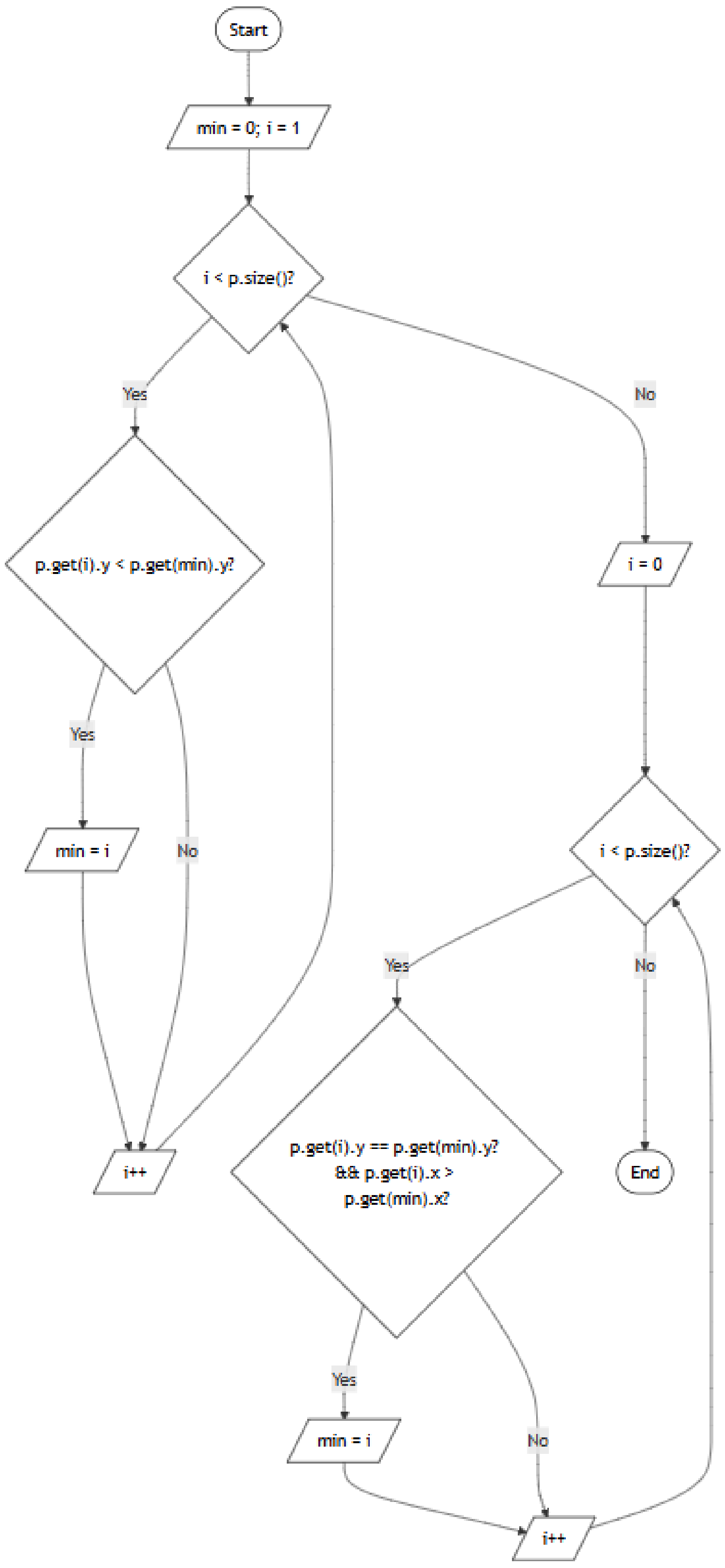
Smit Godhani – 202201162

1. The code below is part of a method in the ConvexHull class in the VMAP system. The following is a small fragment of a method in the ConvexHull class. For the purposes of this exercise, you do not need to know the intended function of the method. The parameter p is a Vector of Point objects, p.size() is the size of the vector p, (p.get(i)).x is the x component of the ith point appearing in p, similarly for (p.get(i)).y. This exercise is concerned with structural testing of code, so the focus is on creating test sets that satisfy some particular coverage criteria.

```java
Vector doGraham(Vector p) {
    int i, j, min, M;
    Point t;
    min = 0;


    // Search for minimum y-coordinate
    for (i = 1; i < p.size(); ++i) {
        if (((Point) p.get(i)).y < ((Point) p.get(min)).y) {
            min = i;
        }
    }
   // Continue along values with the same y component
    for (i = 0; i < p.size(); ++i) {
        if ((((Point) p.get(i)).y == ((Point) p.get(min)).y) &&
            (((Point) p.get(i)).x > ((Point) p.get(min)).x)) {
            min = i;
        }
    }
}
```

Task 1. Convert the code comprising the beginning of the doGraham method into a control flow graph (CFG). You are free to write the code in any programming language.



Task 2: Construct test sets for your flow graph that are adequate for the following criteria:

1. **Statement Coverage:**
   - TC1: p = [(2, 8), (3, 4), (7, 2)]

     Covers cases where p.get(i).y < p.get(min).y is true.
   - TC2: p = [(2, 2), (3, 2), (5, 2), (6, 2)]

     Covers cases where p.get(i).y == p.get(min).y and p.get(i).x > p.get(min).x.
2. **Branch Coverage:**
   - TC3: p = [(3, 4), (6, 6), (7, 10), (11, 16)]

     Covers the false branch for the condition p.get(i).y < p.get(min).y.
   - TC3 (Revised): p = [(11, 16), (7, 10), (6, 6), (3, 4)]

     Covers the true branch for the condition p.get(i).y < p.get(min).y.
   - TC4: p = [(2, 2), (4, 2), (5, 2)]

     Covers both true and false branches of p.get(i).y == p.get(min).y and p.get(i).x > p.get(min).x.
3. **Basic Condition Coverage:**
   - TC5: p = [(3, 5), (5, 6)]

     Covers p.get(i).y < p.get(min).y as false.
   - TC6: p = [(4, 7), (4, 5)]

     Covers p.get(i).y < p.get(min).y as true.
   - TC7: p = [(2, 3), (4, 3)]

     Covers p.get(i).y == p.get(min).y as true and p.get(i).x > p.get(min).x as true.
   - TC8: p = [(1, 1), (2, 2)]

     Covers p.get(i).y == p.get(min).y as false and p.get(i).x > p.get(min).x as false.

Task 3: For the test set you have just checked can you find a mutation of the code (i.e. the deletion, change or insertion of some code) that will result in failure but is not detected by your test set. You have to use the mutation testing tool.

1. **Deletion Mutation**

   Original Code:

```
if ((p.get(i)).y < (p.get(min)).y) {
    min = i;
}
```

   Mutated Code (Condition Check Deleted):

```
min = i;
```

Analysis for Statement Coverage:

- **Effect:** By deleting the condition, the code will always assign i to min, disregarding whether (p.get(i)).y is actually less than (p.get(min)).y. This could lead to incorrect outcomes if min is assigned without validating the smallest y value.
- **Potential Undetected Outcome:** If the test set only verifies that min is assigned without checking if it correctly identifies the point with the minimum y value, this mutation might pass undetected.

2. **Change Mutation**

   Original Code:

```
if ((p.get(i)).y < (p.get(min)).y)
```

   Mutated Code (Condition Check Deleted):

```
if ((p.get(i)).y <= (p.get(min)).y)
```

Analysis for Statement Coverage:

- **Effect:** Changing < to <= means that min will be updated even when p.get(i).y is equal to p.get(min).y. This could lead to incorrectly selecting a point as the minimum when two points have the same y value.
- **Potential Undetected Outcome:** If the test set does not include cases where p.get(i).y equals p.get(min).y, this mutation may go undetected, as there would be no validation against incorrectly updating min for equal y values.

3. **Insertion Mutation**

Original Code:

```
min = i;
```

Mutated Code (Condition Check Deleted):

```
min = i + 1;
```

Analysis for Statement Coverage:

- **Effect:** Adding +1 to i changes the index assigned to min, potentially leading to an out-of-bounds access or incorrect index selection. This mutation deviates from the intended logic of directly assigning i to min.
- **Potential Undetected Outcome:** If the test set only verifies that min is assigned without checking if min points to the expected index, this mutation might remain undetected. Tests need to specifically verify that min holds the correct index without any alterations.

Task 4: Create a test set that satisfies the path coverage criterion where every loop is explored at least zero, one or two times.

```python
import unittest

from point import Point, find_min_point

class TestFindMinPointPathCoverage(unittest.TestCase):

    def test_no_points(self):

        points = []

        with self.assertRaises(IndexError):

            find_min_point(points)

    def test_single_point(self):

        points = [Point(5, 5)]

        result = find_min_point(points)

        self.assertEqual(result, points[0])

    def test_two_points_unique_min(self):

        points = [Point(2, 8), Point(3, 9)]

        result = find_min_point(points)

        self.assertEqual(result, points[0])

    def test_multiple_points_unique_min(self):

        points = [Point(6, 7), Point(3, 6), Point(4, 2)]

        result = find_min_point(points)

        self.assertEqual(result, points[2])

    def test_multiple_points_same_y(self):

        points = [Point(5, 3), Point(6, 3), Point(4, 3)]

        result = find_min_point(points)

        self.assertEqual(result, points[1])

    def test_multiple_points_minimum_y_ties(self):

        points = [Point(8, 4), Point(5, 4), Point(7, 1), Point(9, 1)]

        result = find_min_point(points)

        self.assertEqual(result, points[3])
```

```
if __name__ == "__main__":

    unittest.main()
```

Lab Execution:

1. After generating the control flow graph, check whether your CFG matches with the CFG generated by Control Flow Graph Factory Tool and Eclipse flow graph generator.

Ans. Control Flow Graph Factory :- YES

Eclipse flow graph generator :- YES

2. Devise minimum number of test cases required to cover the code using the aforementioned criteria.

1. Statement Coverage: 3 test cases
2. Branch Coverage: 3 test cases
3. Basic Condition Coverage: 2 test cases
4. Path Coverage: 3 test cases

Summary of Minimum Test Cases:

- Total: 3 (Statement) + 3 (Branch) + 2 (Basic Condition) + 3 (Path) = 11 test cases

Q - 3 and **Q - 4** same as Part 1 which we done above.